9

Generation: Pushing the Specification Level Upwards

> The programmer at wit's end ... can often do best by disentangling himself from his code, rearing back, and contemplating his data. Representation is the essence of programming.
>
> *The Mythical Man-Month, Anniversary Edition (1975–1995), p. 103*
> —Fred Brooks

In Chapter 1 we observed that human beings are better at visualizing data than they are at reasoning about control flow. We recapitulate: To see this, compare the expressiveness and explanatory power of a diagram of a fifty-node pointer tree with a flowchart of a fifty-line program. Or (better) of an array initializer expressing a conversion table with an equivalent switch statement. The difference in transparency and clarity is dramatic.[1]

Data is more tractable than program logic. That's true whether the data is an ordinary table, a declarative markup language, a templating system, or a set of macros that will expand to program logic. It's good practice to move as much of the complexity

1. For further development of this point see [Bentley].

in your design as possible away from procedural code and into data, and good practice to pick data representations that are convenient for humans to maintain and manipulate. Translating those representations into forms that are convenient for machines to process is another job for machines, not for humans.

> Another important advantage of higher-level, more declarative notations is that they lend themselves better to compile-time checking. Procedural notations inherently have complex run-time behavior which is difficult to analyze at compile time. Declarative notations give the implementation much more leverage for finding mistakes, by permitting much more thorough understanding of the intended behavior.
>
> —Henry Spencer

These insights ground in theory a set of practices that have always been an important part of the Unix programmer's toolkit—very high-level languages, data-driven programming, code generators, and domain-specific minilanguages. What unifies these is that they are all ways of lifting the generation of code up some levels, so that specifications can be smaller. We've previously noted that defect densities tend to be nearly constant across programming languages; all these practices mean that whatever malign forces generate our bugs will get fewer lines to wreak their havoc on.

In Chapter 8 we discussed the uses of domain-specific minilanguages. In Chapter 14 we'll make the argument for very-high-level languages. In this chapter we'll look at some design studies in data-driven programming and a few examples of ad-hoc code generation; we'll look at some code-generation tools in Chapter 15. As with minilanguages, these methods can enable you to drastically cut the line count of your programs, and correspondingly lower debugging time and maintenance costs.

9.1 Data-Driven Programming

When doing *data-driven programming*, one clearly distinguishes code from the data structures on which it acts, and designs both so that one can make changes to the logic of the program by editing not the code but the data structure.

Data-driven programming is sometimes confused with object orientation, another style in which data organization is supposed to be central. There are at least two differences. One is that in data-driven programming, the data is not merely the state of some object, but actually defines the control flow of the program. Where the primary concern in OO is encapsulation, the primary concern in data-driven programming is writing as little fixed code as possible. Unix has a stronger tradition of data-driven programming than of OO.

Programming data-driven style is also sometimes confused with writing state machines. It is in fact possible to express the logic of a state machine as a table or data structure, but hand-coded state machines are usually rigid blocks of code that are far harder to modify than a table.

An important rule when doing any kind of code generation or data-driven programming is this: always push problems upstream. Don't hack the generated code or any intermediate representations by hand—instead, think of a way to improve or replace your translation tool. Otherwise you're likely to find that hand-patching bits which should have been generated correctly by machine will have turned into an infinite time sink.

At the upper end of its complexity scale, data-driven programming merges into writing interpreters for p-code or simple minilanguages of the kind we surveyed in Chapter 8. At other edges, it merges into code generation and state-machine programming. The distinctions are not actually that important; the important part is moving program logic away from hardwired control structures and into data.

9.1.1 Case Study: *ascii*

I maintain a program called *ascii*, a very simple little utility that tries to interpret its command-line arguments as names of ASCII characters and report all the equivalent names. Code and documentation for the tool are available from the project page (`http://www.catb.org/~esr/ascii`). Here is an illustrative screenshot:

```
esr@snark:~/WWW/writings/taoup$ ascii 10
ASCII 1/0 is decimal 016, hex 10, octal 020, bits 00010000: called ^P, DLE
Official name: Data Link Escape

ASCII 0/10 is decimal 010, hex 0a, octal 012, bits 00001010: called ^J, LF, NL
Official name: Line Feed
C escape: '\n'
Other names: Newline

ASCII 0/8 is decimal 008, hex 08, octal 010, bits 00001000: called ^H, BS
Official name: Backspace
C escape: '\b'
Other names:

ASCII 0/2 is decimal 002, hex 02, octal 002, bits 00000010: called ^B, STX
Official name: Start of Text
```

One indication that this program was a good idea is the fact that it has an unexpected use—as a quick CLI aid to converting between decimal, hex, octal, and binary representations of bytes.

The main logic of this program could have been coded as a 256-branch case statement. This would, however, have made the code bulky and difficult to maintain. It would also have tangled parts that change relatively rapidly (like the list of slang names for characters) with parts that change slowly or not at all (like the official names), putting them both in the same legend string and making errors during editing much more likely to touch data that ought to be stable.

Instead, we apply data-driven programming. All the character name strings live in a table structure that is quite a bit larger than any of the functions in the code (indeed, counted in lines it is larger than any *three* of the functions in the program). The code merely navigates the table and does low-level tasks like radix conversions. The initializer actually lives in the file `nametable.h`, which is generated in a way we'll describe later in this chapter.

This organization makes it easy to add new character names, change existing ones, or delete old names by simply editing the table, without disturbing the code.

(The way the program is built is good Unix style, but the output format is questionable. It's hard to see how the output could usefully become the input of any other program, so it does not play well with others.)

9.1.2 Case Study: Statistical Spam Filtering

One interesting case of data-driven programming is statistical learning algorithms for detecting spam (unsolicited bulk email). A whole class of mail filter programs (those easily findable by web search include *popfile*, *spambayes*, and *bogofilter*) use a database of word correlations to replace the elaborate pattern-matching conditional logic of pattern-matching spam filters.

Programs like these became common on the Internet very rapidly following Paul Graham's landmark paper *A Plan for Spam* [Graham] in 2002. While the explosion was triggered by the increasing cost of the pattern-matching arms race, the statistical-filtering idea was adopted first and fastest by Unix shops. In part, this was certainly because almost all the Internet service providers (who are most burdened by spam, and thus had most incentive to adopt effective new techniques) are Unix shops—but undoubtedly the harmony with some traditional themes in Unix software design helped as well.

Conventional spam filters require that a system administrator, or some other responsible party, maintain information on patterns of text found in spam—names of sites that emit nothing but spam, come-on phrases often used by pornography sites or Internet scam artists, and the like. In his paper, Graham noted accurately that computer programmers like the idea of pattern-matching filters, and sometimes have difficulty seeing past that approach, because it offers them so many opportunities to be clever.

Statistical spam filters, on the other hand, work by collecting feedback about what the user judges to be spam versus nonspam. That feedback is processed into databases of statistical correlation coefficients or *weights* connecting words or phrases to the user's spam/nonspam classification. The most popular algorithms use minor variants of Bayes's Theorem on conditional probabilities, but other techniques (including various sorts of polynomial hashing) are also employed.

In all these programs, the correlation check is a relatively trivial mathematical formula. The weights fed into the formula along with the message being checked serve as implicit control structure for the filtering algorithm.

The problem with conventional pattern-matching spam filters is that they are brittle. Spammers are constantly gaming against the filter-rule databases, forcing the filter maintainers to constantly reprogram their filters to stay ahead in the arms race. Statistical spam filters generate their own filter rules from the user feedback.

In fact, experience with statistical filters seems to show that the particular learning algorithm used is far less important than the quality of the spam and non-spam data sets from which the learning algorithm computes its weights. So the results of statistical filters really are driven more by the shape of the data than by the algorithm.

A Plan for Spam was something of a bombshell because its author argued convincingly that a simple, even crude, statistical approach gave a lower rate of non-spam being erroneously classified as spam than either elaborate pattern-matching techniques or the human eyeball could manage. For Unix programmers, seeing past the lure of clever pattern-matching was far easier than in other cultures without a strong attachment to "Keep It Simple, Stupid!"

9.1.3 Case Study: Metaclass Hacking in *fetchmailconf*

The *fetchmailconf*(1) dotfile configurator shipped with *fetchmail*(1) contains an instructive example of advanced data-driven programming in a very high-level, object-oriented language.

In October 1997 a series of questions on the fetchmail-friends mailing list made it clear that end-users were having increasing troubles generating configuration files for *fetchmail*. The file uses a simple, classically-Unixy free-format syntax, but can become forbiddingly complicated when a user has POP3 and IMAP accounts at multiple sites. See Example 9.1 for a somewhat simplified version of the *fetchmail* author's configuration file.

The design objective of *fetchmailconf* was to completely hide the control file syntax behind a fashionable, ergonomically-correct GUI interface replete with selection buttons, slider bars and fill-out forms. But the beta design had a problem: it could easily generate configuration files from the user's GUI actions, but could not read and edit existing ones.

Example 9.1: Example of fetchmailrc syntax.

```
set postmaster "esr"
set daemon 300

poll imap.ccil.org with proto IMAP and options no dns
    aka snark.thyrsus.com locke.ccil.org ccil.org
       user esr there is esr here
            options fetchall dropstatus warnings 3600

poll imap.netaxs.com with proto IMAP
       user "esr" there is esr here options dropstatus warnings 3600

skip pop.tems.com with proto POP3:
 user esr here is ed there options fetchall
```

The parser for *fetchmail*'s configuration file syntax is rather elaborate. It's actually written in *yacc* and *lex*, the two classic Unix tools for generating language-parsing code in C. For *fetchmailconf* to be able to edit existing configuration files, it at first appeared that it would be necessary to replicate that elaborate parser in fetchmailconf's implementation language—Python.

This tactic seemed doomed. Even leaving aside the amount of duplicative work implied, it is notoriously hard to be certain that two parsers in two different languages accept the same grammar. Keeping them synchronized as the configuration language evolved bid fair to be a maintenance nightmare. It would have violated the SPOT rule we discussed in Chapter 4 wholesale.

This problem stumped me for a while. The insight that cracked it was that *fetchmailconf* could use *fetchmail*'s own parser as a filter! I added a `--configdump` option to *fetchmail* that would parse `.fetchmailrc` and dump the result to standard output in the format of a Python initializer. For the file above, the result would look roughly like Example 9.2 (to save space, some data not relevant to the example is omitted).

The major hurdle had been leapt. The Python interpreter could then evaluate the *fetchmail* `--configdump` output and read the configuration available to *fetchmailconf* as the value of the variable 'fetchmail'.

But this wasn't quite the last obstacle in the race. What was really needed wasn't just for *fetchmailconf* to have the existing configuration, but to turn it into a linked tree of live objects. There would be three kinds of object in this tree: `Configuration` (the top-level object representing the entire configuration), `Site` (representing one of the servers to be polled), and `User` (representing user data attached to a site). The example file describes three site objects, each with one user object attached to it.

Example 9.2: Python structure dump of a *fetchmail* configuration.

```
fetchmailrc = {
    'poll_interval':300,
    "logfile":None,
    "postmaster":"esr",
    'bouncemail':TRUE,
    "properties":None,
    'invisible':FALSE,
    'syslog':FALSE,
    # List of server entries begins here
    'servers': [
 # Entry for site `imap.ccil.org' begins:
 {
     "pollname":"imap.ccil.org",
     'active':TRUE,
     "via":None,
     "protocol":"IMAP",
     'port':0,
     'timeout':300,
     'dns':FALSE,
     "aka":["snark.thyrsus.com","locke.ccil.org","ccil.org"],
     'users': [
  {
      "remote":"esr",
      "password":"Malvern",
      'localnames':["esr"],
      'fetchall':TRUE,
      'keep':FALSE,
      'flush':FALSE,
      "mda":None,
      'limit':0,
      'warnings':3600,
  }
  ,     ]
 }
 ,
 # Entry for site `imap.netaxs.com' begins:
 {
     "pollname":"imap.netaxs.com",
     'active':TRUE,
     "via":None,
     "protocol":"IMAP",
     'port':0,
     'timeout':300,
     'dns':TRUE,
     "aka":None,
     'users': [
```

```
  {
      "remote":"esr",
      "password":"d0wnthere",
      'localnames':["esr"],
      'fetchall':FALSE,
      'keep':FALSE,
      'flush':FALSE,
      "mda":None,
      'limit':0,
      'warnings':3600,
  }
  ,     ]
 }
 ,
 # Entry for site `pop.tems.com' begins:
 {
     "pollname":"pop.tems.com",
     'active':FALSE,
     "via":None,
     "protocol":"POP3",
     'port':0,
     'timeout':300,
     'dns':TRUE,
     'uidl':FALSE,
     "aka":None,
     'users': [
  {
      "remote":"ed",
      "password":None,
      'localnames':["esr"],
      'fetchall':TRUE,
      'keep':FALSE,
      'flush':FALSE,
      "mda":None,
      'limit':0,
      'warnings':3600,
  }
  ,     ]
 }
    ]
}
```

The three object classes already existed in *fetchmailconf*. Each had a method that caused it to pop up a GUI edit panel to modify its instance data. The last remaining problem was to somehow transform the static data in this Python initializer into live objects.

I considered writing a glue layer that would explicitly know about the structure of all three classes and use that knowledge to grovel through the initializer creating matching objects, but rejected that idea because new class members were likely to be added over time as the configuration language grew new features. If the object-creation code were written in the obvious way, it would once again be fragile and tend to fall out of synchronization when either the class definitions or the initializer structure dumped by the `--configdump` report generator changed. Again, a recipe for endless bugs.

The better way would be data-driven programming—code that would analyze the shape and members of the initializer, query the class definitions themselves about their members, and then impedance-match the two sets.

Lisp and Java programmers call this *introspection*; in object-oriented languages it's called *metaclass hacking* and is generally considered fearsomely esoteric, deep black magic. Most object-oriented languages don't support it at all; in those that do (Perl and Java among them), it tends to be a complicated and fragile undertaking. But Python's facilities for introspection and metaclass hacking are unusually accessible.

See Example 9.3 for the solution code, from near line 1895 of the 1.43 version.

Most of this code is error-checking against the possibility that the class members and `--configdump` report generation have drifted out of synchronization. It ensures that if the code breaks, the breakage will be detected early—an implementation of the Rule of Repair. The heart of this function is the last two lines, which set attributes in the class from corresponding members in the dictionary. They're equivalent to this:

```
def copy_instance(toclass, fromdict):
        for x in fromdict.keys():
                setattr(toclass, x, fromdict[x])
```

When your code is this simple, it is far more likely to be right. See Example 9.4 for the code that calls it.

The key point to extract from this code is that it traverses the three levels of the initializer (configuration/server/user), instantiating the correct objects at each level into lists contained in the next object up. Because `copy_instance` is data-driven and completely generic, it can be used on all three levels for three different object types.

Example 9.3: `copy_instance` metaclass code.

```
def copy_instance(toclass, fromdict):
# Make a class object of given type from a conformant dictionary.
    class_sig = toclass.__dict__.keys(); class_sig.sort()
    dict_keys = fromdict.keys(); dict_keys.sort()
    common = set_intersection(class_sig, dict_keys)
    if 'typemap' in class_sig:
 class_sig.remove('typemap')
    if tuple(class_sig) != tuple(dict_keys):
 print "Conformability error"
# print "Class signature: " + `class_sig`
# print "Dictionary keys: " + `dict_keys`
 print "Not matched in class signature: "+ \
                                        `set_diff(class_sig, common)`
 print "Not matched in dictionary keys: "+ \
                                        `set_diff(dict_keys, common)`
 sys.exit(1)
    else:
 for x in dict_keys:
     setattr(toclass, x, fromdict[x])
```

This is a new-school sort of example; Python was not even invented until 1990. But it reflects themes that go back to 1969 in the Unix tradition. If meditating on Unix programming as practiced by his predecessors had not taught me constructive laziness—insisting on reuse, and refusing to write duplicative glue code in accordance with the SPOT rule—I might have rushed into coding a parser in Python. The first key insight that *fetchmail* itself could be made into *fetchmailconf*'s configuration parser might never have happened.

The second insight (that `copy_instance` could be generic) proceeded from the Unix tradition of looking assiduously for ways to avoid hand-hacking. But more specifically, Unix programmers are very used to writing parser specifications to generate parsers for processing language-like markups; from there it was a short step to believing that the rest of the job could be done by some kind of generic tree-walk of the configuration structure. Two separate stages of data-driven programming, one building on the other, were needed to solve the design problem cleanly.

Insights like this can be extraordinarily powerful. The code we have been looking at was written in about ninety minutes, worked the first time it was run, and has been stable in the years since (the only time it has ever broken is when it threw an exception in the presence of genuine version skew). It's less than forty lines and beautifully simple. There is no way that the naïve approach of building an entire second parser could possibly have produced this kind of maintainability, reliability or compactness. Re-use, simplification, generalization, orthogonality: this is the Zen of Unix in action.

Example 9.4: Calling context for `copy_instance`.

```
    # The tricky part - initializing objects from the `configuration'
    # global.  `Configuration' is the top level of the object tree
    # we're going to mung
    Configuration = Controls()
    copy_instance(Configuration, configuration)
    Configuration.servers = [];
    for server in configuration['servers']:
Newsite = Server()
copy_instance(Newsite, server)
Configuration.servers.append(Newsite)
Newsite.users = [];
for user in server['users']:
    Newuser = User()
    copy_instance(Newuser, user)
    Newsite.users.append(Newuser)
```

In Chapter 10, we'll examine the run-control syntax of *fetchmail* as an example of the standard shell-like metaformat for run-control files. In Chapter 14 we'll use *fetchmailconf* as an example of Python's strength in rapidly building GUI interfaces.

9.2 Ad-hoc Code Generation

Unix comes equipped with some powerful special-purpose code generators for purposes like building lexical analyzers (tokenizers) and parsers; we'll survey these in Chapter 15. But there are much simpler, lighter-weight sorts of code generation we can use to make life easier without having to know any compiler theory or write (error-prone) procedural logic.

Here are a couple of simple case studies to illustrate this point:

9.2.1 Case Study: Generating Code for the *ascii* Displays

Called without arguments, *ascii* generates a usage screen that looks like Example 9.5.

This screen is carefully designed to fit in 23 rows and 79 columns, so that it will fit in a 24×80 terminal window.

This table could be generated at runtime, on the fly. Grinding out the decimal and hex columns would be easy enough. But between wrapping the table at the right places and knowing when to print mnemonics like NUL rather than characters, there would have been enough odd corner cases to make the code distinctly unpleasant. Furthermore, the columns had to be unevenly spaced to make the table fit in 79 columns. But

Example 9.5: *ascii* usage screen.

```
Usage: ascii [-dxohv] [-t] [char-alias...]
   -t = one-line output  -d = Decimal table  -o = octal table  -x = hex table
   -h = This help screen -v = version information
Prints all aliases of an ASCII character. Args may be chars, C \-escapes,
English names, ^-escapes, ASCII mnemonics, or numerics in decimal/octal/hex.

Dec Hex    Dec Hex    Dec Hex  Dec Hex  Dec Hex  Dec Hex   Dec Hex   Dec Hex
  0 00 NUL  16 10 DLE  32 20    48 30 0  64 40 @  80 50 P   96 60 `  112 70 p
  1 01 SOH  17 11 DC1  33 21 !  49 31 1  65 41 A  81 51 Q   97 61 a  113 71 q
  2 02 STX  18 12 DC2  34 22 "  50 32 2  66 42 B  82 52 R   98 62 b  114 72 r
  3 03 ETX  19 13 DC3  35 23 #  51 33 3  67 43 C  83 53 S   99 63 c  115 73 s
  4 04 EOT  20 14 DC4  36 24 $  52 34 4  68 44 D  84 54 T  100 64 d  116 74 t
  5 05 ENQ  21 15 NAK  37 25 %  53 35 5  69 45 E  85 55 U  101 65 e  117 75 u
  6 06 ACK  22 16 SYN  38 26 &  54 36 6  70 46 F  86 56 V  102 66 f  118 76 v
  7 07 BEL  23 17 ETB  39 27 '  55 37 7  71 47 G  87 57 W  103 67 g  119 77 w
  8 08 BS   24 18 CAN  40 28 (  56 38 8  72 48 H  88 58 X  104 68 h  120 78 x
  9 09 HT   25 19 EM   41 29 )  57 39 9  73 49 I  89 59 Y  105 69 i  121 79 y
 10 0A LF   26 1A SUB  42 2A *  58 3A :  74 4A J  90 5A Z  106 6A j  122 7A z
 11 0B VT   27 1B ESC  43 2B +  59 3B ;  75 4B K  91 5B [  107 6B k  123 7B {
 12 0C FF   28 1C FS   44 2C ,  60 3C <  76 4C L  92 5C \  108 6C l  124 7C |
 13 0D CR   29 1D GS   45 2D -  61 3D =  77 4D M  93 5D ]  109 6D m  125 7D }
 14 0E SO   30 1E RS   46 2E .  62 3E >  78 4E N  94 5E ^  110 6E n  126 7E ~
 15 0F SI   31 1F US   47 2F /  63 3F ?  79 4F O  95 5F _  111 6F o  127 7F DEL
```

any Unix programmer would reflexively express it as a block of data before finding out these things.

The most naïve way to generate the usage screen would have been to put each line into a C initializer in the `ascii.c` source code, and then have all lines be written out by code that steps through the initializer. The problem with this method is that the extra data in the C initializer format (trailing newline, string quotes, comma) would make the lines longer than 79 characters, causing them to wrap and making it rather difficult to map the appearance of the code to the appearance of the output. This, in turn, would make the display difficult to edit, which was annoying when I was tinkering it to fit in 24×80 screen cells.

A more sophisticated method using the string-pasting behavior of the ANSI C preprocessor collided with a variant of the same problem. Essentially, any way of inlining the usage screen explicitly would involve punctuation at start and end of line

that there's no room for.[2] And copying the table to the screen from a file at runtime seemed like a fragile expedient; after all, the file could get lost.

Here's the solution. The source distribution contains a file that just contains the usage screen, exactly as listed above and named `splashscreen`. The C source contains the following function:

```
void
showHelp(FILE *out, char *progname)
{
  fprintf(out,"Usage: %s [-dxohv] [-t] [char-alias...]\n", progname);
#include "splashscreen.h"

  exit(0);
}
```

And `splashscreen.h` is generated by a makefile production:

```
splashscreen.h: splashscreen
 sed <splashscreen >splashscreen.h \
            -e 's/\\/\\\\/g' -e 's/"/\\"/' -e 's/.*/puts("&");/'
```

So when the program is built, the `splashscreen` file is automatically massaged into a series of output function calls, which are then included by the C preprocessor in the right function.

By generating the code from data, we get to keep the editable version of the usage screen identical to its display appearance. This promotes transparency. Furthermore, we could modify the usage screen at will without touching the C code at all, and the right thing would automatically happen on the next build.

For similar reasons, the initializer that holds the name synonym strings is also generated via a *sed* script in the makefile, from a file called `nametable` in the ascii source distribution. Most of `nametable` is simply copied into the C initializer. But the generation process would make it easy to adapt this tool for other 8-bit character sets such as the ISO-8859 series (Latin-1 and friends).

This is an almost trivial example, but it nevertheless illustrates the advantages of even simple and ad-hoc code generation. Similar techniques could be applied to larger programs with correspondingly greater benefits.

2. Scripting languages tend to solve this problem more elegantly than C does. Investigate the shell's *here documents* and Python's triple-quote construct to find out how.

9.2.2 Case Study: Generating HTML Code for a Tabular List

Let's suppose that we want to put a page of tabular data on a Web page. We want the first few lines to look like Example 9.6.

Example 9.6: Desired output format for the star table.

```
Aalat          David Weber                The Armageddon Inheritance
Aelmos         Alan Dean Foster           The Man who Used the Universe
Aedryr         Steve Miller/Sharon Lee    Scout's Progress
Aergistal      Gerard Klein               The Overlords of War
Afdiar         L. Neil Smith              Tom Paine Maru
Agandar        Donald Kingsbury           Psychohistorical Crisis
Aghirnamirr    Jo Clayton                 Shadowkill
```

The thick-as-a-plank way to handle this would be to hand-write HTML table code for the desired appearance. Then, each time we want to add a name, we'd have to hand-write another set of `<tr>` and `<td>` tags for the entry. This would get very tedious very quickly. But what's worse, changing the format of the list would require hand-hacking every entry.

The superficially clever way to handle this would be to make this data a three-column relation in a database, then use some fancy CGI technique or a database-capable templating engine like PHP to generate the page on the fly. But suppose we know that the list will not change very often, don't want to run a database server just to be able to display this list, and don't want to load the server with unnecessary CGI traffic?

There's a better solution. We put the data in a tabular flat-file format like Example 9.7.

Example 9.7: Master form of the star table.

```
Aalat         :David Weber               :The Armageddon Inheritance
Aelmos        :Alan Dean Foster          :The Man who Used the Universe
Aedryr        :Steve Miller/Sharon Lee   :Scout's Progress
Aergistal     :Gerard Klein              :The Overlords of War
Afdiar        :L. Neil Smith             :Tom Paine Maru
Agandar       :Donald Kingsbury          :Psychohistorical Crisis
Aghirnamirr   :Jo Clayton                :Shadowkill
```

We could in a pinch have done without the explicit colon field delimiters, using the pattern consisting of two or more spaces as a delimiter, but the explicit delimiter protects us in case we press spacebar twice while editing a field value and fail to notice it.

We then write a script in shell, Perl, Python, or Tcl that massages this file into an HTML table, and run that each time we add an entry. The old-school Unix way would revolve around the following nigh-unreadable *sed*(1) invocation

```
sed -e 's,^,<tr><td>,' -e 's,$,</td></tr>,' -e 's,:,</td><td>,g'
```

or this perhaps slightly more scrutable *awk*(1) program:

```
awk -F: '{printf("<tr><td>%s</td><td>%s</td><td>%s</td></tr>\n", \
                 $1, $2, $3)}'
```

(If either of these examples interests but mystifies, read the documentation for *sed*(1) or *awk*(1). We explained in Chapter 8 that the latter has largely fallen out of use. The former is still an important Unix tool that we haven't examined in detail because (a) Unix programmers already know it, and (b) it's easy for non-Unix programmers to pick up from the manual page once they grasp the basic ideas about pipelines and redirection.)

A new-school solution might center on this Python code, or on equivalent Perl:

```
for row in map(lambda x:x.rstrip().split(':'),sys.stdin.readlines()):
    print "<tr><td>" + "</td><td>".join(row) + "</td></tr>"
```

These scripts took about five minutes each to write and debug, certainly less time than would have been required to either hand-hack the initial HTML or create and verify the database. The combination of the table and this code will be much simpler to maintain than either the under-engineered hand-hacked HTML or the over-engineered database.

A further advantage of this way of solving the problem is that the master file stays easy to search and modify with an ordinary text editor. Another is that we can experiment with different table-to-HTML transformations by tweaking the generator script, or easily make a subset of the report by putting a *grep*(1) filter before it.

I actually use this technique to maintain the Web page that lists *fetchmail* test sites; the example above is science-fictional only because publishing the real data would reveal account usernames and passwords.

This was a somewhat less trivial example than the previous one. What we've actually designed here is a separation between content and formatting, with the generator script acting as a stylesheet. (This is yet another mechanism-vs.-policy separation.)

The lesson in all these cases is the same. Do as little work as possible. Let the data shape the code. Lean on your tools. Separate mechanism from policy. Expert Unix programmers learn to see possibilities like these quickly and automatically. Constructive laziness is one of the cardinal virtues of the master programmer.